Compiler-Driven Power Optimizations in the Register File of Processor-Based Systems

José L. Ayala and Marisa López-Vallejo

Universidad Politécnica de Madrid, Departamento de Ingeniería Electrónica
Ciudad Universitaria s/n, 28040 Madrid, Spain
{jayala,marisa}@die.upm.es

Abstract. Continuing advances in semiconductor technology have allowed dramatic performance gains for general-purpose microprocessors and embedded systems with the fact that power consumption is also increased. Moreover, a new problem arises because the power savings achievable with low level techniques are reaching their theoretical maximum. Therefore, high level techniques to reduce the power consumption in processor-based systems are needed.
In this context, this work proposes different power-saving techniques for the register file of processor-based systems by means of a power-aware compiler. These techniques modify the compiler optimizations to improve the behavior of a supporting hardware which reduces the energy consumption without performance penalty. This paper also shows the ongoing work on multi-processor systems on-a-chip, which extend the target of the proposed power optimization mechanisms.

Keywords. Power consumption, register file, out-of-order, MPSoC, compiler support, power-aware.

1 Introduction

Processor-based systems represent nowadays a significant fraction of the semiconductor market. Several indicators support this fact. First, applications executed by microprocessor-based systems were related to scientific processing applications. Currently, these applications have reached the user market by executing complex applications in movile cell-phones, digital cameras, movile computers and PDAs, devices for handicapped or cars. Second, the complexity of the scaling, in terms of number of integrated transistors in the silicon die, has been doubled every two years following the Moore's Law. The predictions for the next few years keep close to this trend. Also, in terms of working frequency, the lead microprocessors frequency doubles every two years.

This trend in the extension and increasing complexity of the processor-based systems is followed by an increase on the power dissipation of these devices. The power dissipation of current architectures (like Pentium IV) is over the ten kilowatts. The expectation of tens of kilowatts dissipated in the next five years makes

Dagstuhl Seminar Proceedings 05141
Power-aware Computing Systems
http://drops.dagstuhl.de/opus/volltexte/2005/305

the power delivery and dissipation of these systems to be prohibitive. Moreover, the power density becomes too high to keep junctions at low temperature.

In current embedded and high-performance processors, the register file consumes a sizable fraction of the total power and becomes a dominant source of energy dissipation when other power saving mechanisms have been applied. Register file power consumption depends very much on the system configuration, mainly on the number of integrated registers, cache size and existence of a branch predictor table (i.e. depends on the relative size of other memory devices). In the Motorola's M.CORE architecture, the register file energy consumption could achieve 16% of the total processor power and 42% of the data path power [1]. In out-of-order processors with a large number of physical registers which are implemented as part of the *Re-Order Buffer* (in Pentium III, for example), this structure dissipates as much as 27% of total energy according to some estimates [2].

Moreover, many recent compiler optimizations increase the register pressure, and there is a current trend towards implementing larger register files. Large register files present several advantages: decreased power consumption in the memory hierarchy (cache and main memory) by eliminating accesses, improved performance by decreasing path length and reduced memory traffic by removing load and store operations. However, current research in this area [4] and the efforts on optimizing spill code indicate a need for more registers. Sophisticated optimizations can increase the number of variables and the register pressure; furthermore, global variable register allocation can increase the number of required registers. The number of registers, the register pressure and the aggressive compiler optimizations (by allocating global variables, promoting aliased variables and inlining) will lead to increase the energy consumption.

As was said before, the size of the register file in embedded and high performance processors has shown a significan increase and this motivates the need for efficient techniques to reduce register file energy consumption in embedded systems. This paper is based on a technique which is characterized by an absence of performance penalty. It is based on the observation that a register is only used when an instruction reads from it or writes to it, the register is "idle" at all other times. By keeping the idle registers in a low power (or "drowsy") state [5] a significant amount of energy can be saved. Most registers are idle in any given cycle since at most three registers are accessed by an issued instruction. The compiler and architectural modifications we propose in this paper allow the drowsy registers to be turned back to the "active" state as the instruction accesses them.

Compiler-techology traditionally has focused on improving the program size, quality and performance of the compiled code. However, not many approaches have targeted the power optimizations. Also, the majority of compiler driven approaches to power reduction have focused on embedded processors, while there is not so much work on superscalar out-of-order processors.

In this paper we present two different techniques to reduce the power consumption of the register file in out-of-order architectures by means of a power-

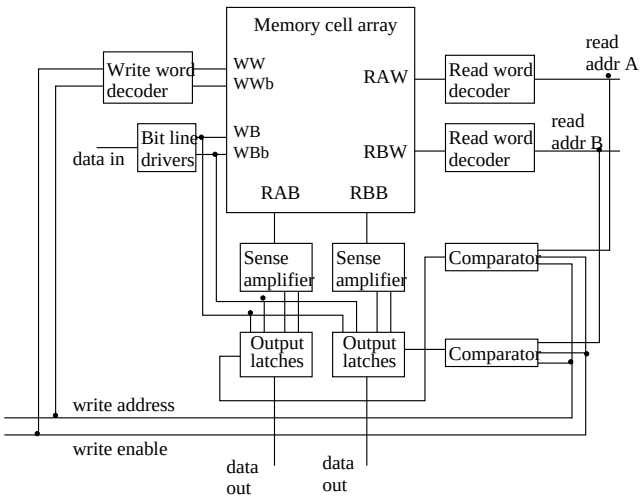

Fig. 1. Register file.

aware compiler. The first technique allows a reduction in the number of ports of the register file by modifying the assignment of registers (and saving energy in this way). The second technique focuses on optimizing the loop unrolling mechanism of the compiler. This work is currently being extended to multi-processor environments, where the defined power saving techniques find a very promising room.

The rest of the paper is organized as follows: section 2 explains the foundations of the approach, while related work is summarized in 3. The techniques to reduce the number of ports and the modification of the loop unrolling mechanism are explained in sections 4 and 5 respectively. Our ongoing work on MPSoCs is shown in section 6 and section 7 finishes the paper with some conclusions.

2 Foundations

In a typical configuration, the register file is an array of N words by M bits. Any of the N words can be simultaneously accessed by two read ports and a single write port. A block diagram of the register file, shown in Figure 1, shows that the register file contains seven distinct types of functional blocks [6]. These are the memory cell array, the read address decoders and word line drivers, the write address decoder, the bit line drivers, the sense amplifiers, the output latches, and the comparators.

The memory cell array stores the bits of data. When any of the ports accesses the memory cell array, the read or write operation is performed on every memory cell in the selected row simultaneously. Each read address decoder is responsible for decoding a $log_2 N$-bit address to determine which of the rows is selected for each read operation. The read word line drivers are responsible for driving the

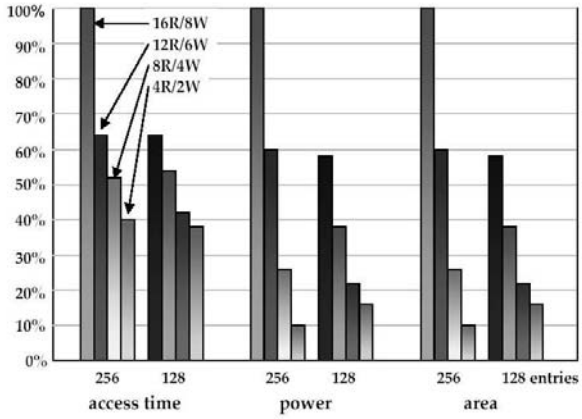

Fig. 2. Effects of size and number of ports on access time, power and area

read word lines accordingly. The write address decoder selects the row to be written. The write word line drivers drive the write word lines while the bit line drivers provide input data to the memory cells.

The independent read and write address decoders allow parallel access to up to three register operands of an instruction. Also, some registers in the register file will never be used due to the locality of data in the code. Therefore, the register file is underutilized most of the time, and this fact will be used to keep those free registers in a low-power state which saves energy [5].

More in detail, the main goals of our research work in this area are to provide the architectural extensions and compiler support to reduce the power consumption in the register file of processor-based systems by reducing the number of ports of this device and by modifying the loop unrolling mechanism. Finally, these ideas will be extended to the multi-processor systems-on-a-chip.

3 Related Work

Most of the previous work on the register file has been related to general size reduction techniques. Some of the authors propose distributed schemes as opposed to a central implementation [7] as well as split microarchitectures into distributed clusters [8].

Other interesting works on reducing the size of the register file propose multi-level implementations [9] or interleaved banks [10]. In [11], the authors also apply voltage scaling techniques to reduce the power consumption when the register file is not accessed.

Compiler optimization mechanisms have been proposed to reduce power consumption in microprocessors [12], but these approaches only work on code optimizations and do not use hardware support. Also, there are solutions based

on code versioning and selection by the compiler using heuristics and profile data [13]. The compiler approaches proposed in this work will be also supported with hardware modifications which improve the results in terms of applicability and energy consumption.

Compiler optimizations such as power-aware instruction selection, i.e., choosing the instruction sequence that will cause the lowest energy dissipation when the program is executed, will not require changes in the hardware to reduce energy consumption [14]. Often, optimizing for performance will result in a program that also yields good energy results. There are cases, however, where these two goals will result in different instruction sequences. The compiler optimization technique register pipelining is an example for this effect [15]. Previous works have also targeted the register file optimization by means of compiler-based techniques such as software pipelining [16], variations to the register renaming task [17], or architecture reconfiguration [18].

Therefore, as was previously said, our work will consider both energy consumption and performance as target variables to be optimized with the proposed techniques.

4 Reduction of Number of Ports in the Register File

It is well known that the number of ports and the size of the register file impact on its energy consumption. Figure 2 shows the effects of size and number of ports on access time, energy and area for 128-, and 256-entry register files having various combinations of read and write ports [19]. The numbers were calculated using a modified version of CACTI 3.0 [20] assuming a 0.18 μm technology. The authors consider four configurations of the register file: one with 16-read and 8-write ports; another with 12-read and 6-write ports; other configuration with 8-read and 4-write ports; and a last one with 4-read and 2-write ports. These are intended to support 8-, 6-, 4-, and 2-issue machines respectively. These values are normalized against a 256-entry register file with 16R/8W ports. Figure 2 illustrates quite dramatically the penalty paid in access time, energy and area as the number of ports is increased.

Our approach splits the register file into independent banks with a reduced number of ports per bank. Those banks that are not required by any operand coded in the instruction word are kept in a low-power state to save as much energy as possible [21]. The energy savings in this approach are due to the voltage scaling applied and the reduced number of ports enabled in every access to the register file.

The architectural modifications presented in this approach are also supported by a power-aware compiler where the register assignment has been modified in order to efficiently use those banks.

The compiler selected to perform the implementation of the improved register assignment policy is the GNU compiler gcc 3.2 which is publicly available, generates high quality code and supports multiple embedded and high-performance processors.

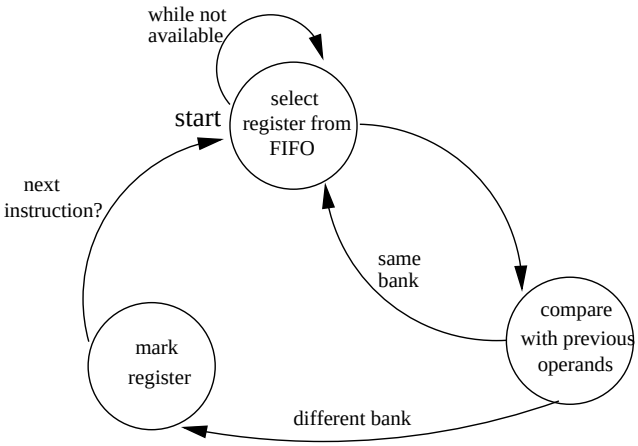

Fig. 3. Register assignment algorithm

Gcc, when assigning an architectural register to the instruction operands, retrieves the first available register from a list (a FIFO) of free registers. The order of the registers inside the list is not representative and depends on the specific hardware architecture. Since gcc does not consider any restriction on assigning the registers, these are selected consecutively and, therefore, all the operands in our approach would come from the same register file bank if no modification to the register assignment algorithm is accomplished.

The register assignment policy implemented in the compiler modifies the traditional assignment by promoting every operand in the instruction to a different register file bank. With this modification, the number of ports in every register file bank is reduced since less accesses per bank are performed in parallel.

The algorithm followed by the compiler to assign the architectural registers is shown in Figure 3. First, the first available register in the list of free registers is selected. This register is double-checked to be free and not system-reserved and, after that, compared to the registers assigned to the other operands of the instruction. If the register file bank for the operand under assignment matches any of the other operands of the instruction, this register is promoted to the next bank and the procedure is repeated. When the register is selected, the liveness of the register is calculated and the annotation is generated.

After the register assignment is completed, it results that every operand of the instruction has been disposed in a different register file bank.

4.1 Experimental Results

The architecture that will be described in the following paragraphs, as well as the compiler modifications, have been implemented and simulated to verify the correctness of the approach and the energy savings obtained. The SimpleScalar's version used in our experiments is 4.0 because this later version includes an

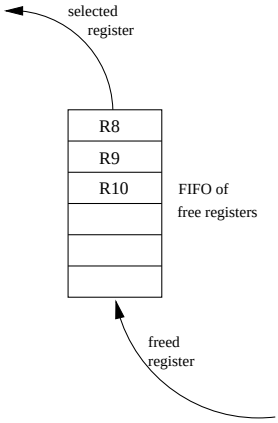

Fig. 4. FIFO of free registers

optimized implementation of the register renaming algorithm for out-of-order processors.

The simulated baseline configuration counts with a multi-ported register file implementation with 256 registers and 22R/11W ports (as found in many implementations, like x86) and it is configured as a 3-issue machine. The benchmarks used in the experiments come from the MediaBench suite.

Analysis of the Number of Banks It is clear that increasing the number of sub-banks of the register file, higher energy savings are expected since a bigger portion of the register file can be kept into the low-power state when it is not accessed. However, there are two main limitations to this idea. First, sometimes when increasing the number of sub-banks from one configuration to another, the number of ports per bank has to be kept constant to meet the performance constraints and avoid pipeline stalls. The energy per access, which is related to the number of ports, does not decrease and the energy savings also decrease due to the increased number of banks. Therefore, some energy penalty can appear when increasing the number of banks.

Second, the alive registers required by the application before any of them can be freed (*liveness*). When the compiler assigns registers during the register allocation phase, it takes them from a list (or FIFO) of free registers, and gives them back to the list when the live period of the register has finished and it can be freed (see Figure 4). If the compiler does not find a free register in the list it has to insert no-operation cycles waiting for available registers.

The analysis of the liveness of the registers assigned by the compiler shows which is the minimum size of the sub-banks to assure that they will never run out of free registers. This analysis is performed as a previous phase to the compilation with the unmodified compiler (the register assignment policy is not modified),

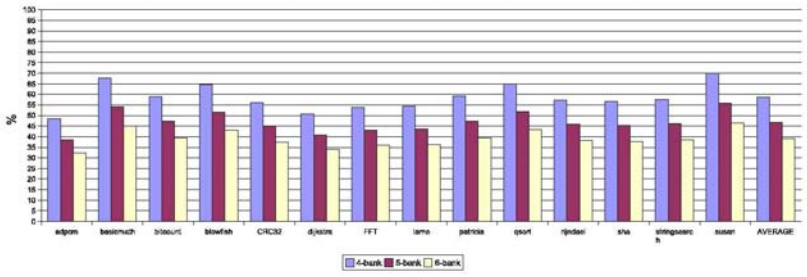

Fig. 5. Energy consumption for the different banks configuration

and can be dumped into a text file containing the liveness of every register and the beginning and ending instruction.

The presented analysis has been performed for the whole set of benchmarks (from the MediaBench suite) used in the simulations, showing that the maximum number of banks that meets the performance constraints in every benchmark is six. Thus, two new configurations have been implemented and simulated to obtain the energy savings: *5-bank configuration* and *6-bank configuration*.

Simulations Attending to the impact of the number of ports in the energy consumption, for the *5-bank configuration* the total energy consumption of the register file is reduced to a **72.37%** of the baseline configuration when splitting the register file into five banks. The *6-bank configuration* shows an extra decrease on these savings (the total energy consumption is reduced to a **66.32%** of the baseline configuration). It should be noticed that configurations with more banks could not improve these numbers. For example, the 7-bank configuration reduces the energy consumption to a **86.84%** because the number of required ports per bank has to be kept equal to the 6-bank configuration (**6** ports per bank in both cases). This fact reflects the first limitation on selecting the number of sub-banks, as was previously explained. Also, it has to be noticed how the complexity of the implementation and the access time to the register file have been dramatically reduced with these last two configurations.

Equally to the *4-bank configuration*, the low-power policy is applied later to obtain greater energy savings. Figure 5 shows the resulting energy consumption of the register file for the three analyzed configurations when the low-power policy is applied. As can be observed, the average energy consumption is heavily decreased in both configurations since a bigger portion of the register file can be turn into the low-power state when it is not required for any instruction. Those benchmarks with less energy consumption correspond to those with more instructions requiring only one operand, while the consumption is increased when more multi-operand instructions are executed.

When increasing the number of register file banks (and thus, when decreasing the number of read and write ports) the pipeline could stall if the issue-width of the machine can not be fed by a sufficient number of register file ports. In

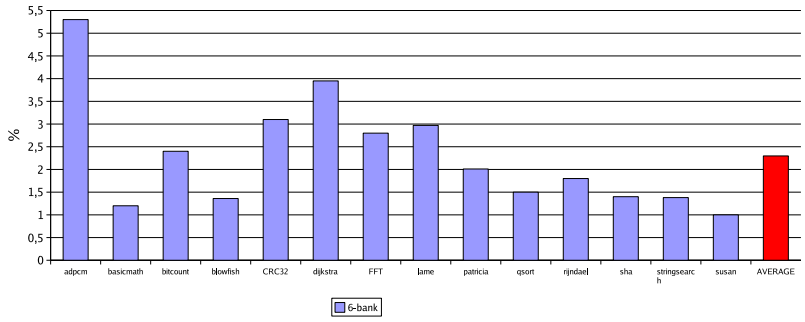

Fig. 6. Performance penalty of the 6-bank register file

the selected configurations, such behavior can be observed for the 6-bank configuration (Figure 6). As can be noticed, the performance penalty shown for the 6-bank configuration (2.3% on average) is in accordance with previous published works. [22,23]

5 Optimizations of the Loop-Unroller

Loop unrolling intends to increase instruction level parallelism of loop bodies by unrolling the loop body multiple times in order to schedule several loop iterations together. The transformation also reduces the number of times loop control statements are executed. As loop unrolling reduces execution time through effective exploitation of ILP from different iterations, it has been presented in the past as an effective compiler mechanism to reduce the energy consumption.

However, loop unrollers perform better for in-order architectures. Current widely-available compilers are not able to exploit the dynamic scheduling facilities found in out-of-order processors, and the ILP improvements are not so spectacular. On the other hand, the unroll of outer loops (or the unroll of inner loops by large factors) exploits the register requirements and increases the energy consumption on the register file. Recent research in modern architectures has shown how loop unrolling proved to have little effect in terms of program execution time [24]. Moreover, these works did not consider the increment on energy consumption due to the increased register usage when the unrolling takes place.

This section presents a power-aware unroller mechanism to efficiently reduce the energy consumption in the register file of out-of-order processors. The modified unroller considers the following alternatives:

- Selection of an unroll factor which fits the register requirements into the available register file bank.
- Use of a dedicated unroll bank of registers to perform unrolling in a safe, energy-controlled space.
- Deactivation of the loop unrolling optimization.

5.1 Selection of the Unroll Factor

Once register assignment is performed by the compiler and before any loop unroll has taken place, the number of required registers inside the ness is perfectly known. The loop unroller exploits the register requirements by placing several copies of the same code and increasing in this way the register demands. Subsequent compiler optimizations (for example, software pipelining) will reduce the number of demanded registers by promoting unused registers or reusing operands.

Therefore, even though the exact number of required registers cannot be known in advance, this number can be estimated. Gcc performs an initial estimate of this parameter based on a worst-case estimation, and such number can be retrived from the gcc's source code and used for our purposes. Also, a post-compilation phase can be considered to tune the unroll factor regarding the effective factor used during the first compilation.

Once the estimated number of required registers is known, the unroller can select the unroll factor which fits the register requirements into the available register file bank while the others remain off.

5.2 Use of the Unroll Bank of Registers

The previous phase can result in the selection of an unroll factor too small if the loop requires a great amount of registers. This reduced unroll factor could determine a penalty in the system performance because other optimizations such as common-subexpresion elimination, induction-variable optimizations, instruction scheduling or software pipelining loose effectiveness. For that reason, an *unroll bank* of registers will be considered.

This unroll bank of registers consists of a bank with reserved registers, whose size is bigger than any of the register file banks, and that remains off during normal execution. When needed, the unroll bank has to be explicitly turned on by the compiler, switching off the rest of register file banks (i.e. the working register file bank is reconfigured to a bigger one to be used during the loop).

In order to perform this operation, the registers currently used as inputs inside the loop have to be moved to the unroll bank of registers (the contents have to be copied). Also, when the loop exits, the output registers have to be moved to the register file bank they belong to. This requires some extra clock cycles to perform the operation, which negatively impact the system performance. Therefore, this compiler optimization will only be allowed in long and frequently executed loops with strong register requirements (i.e. those loops which represent higher energy savings and whose energy-execution trade-off is justified).

Clearly, there is a performance and area overhead due to the use of the extra bank, and the movement of registers from the register file. The section 5.4 will show how this overhead is negligible in a practical case.

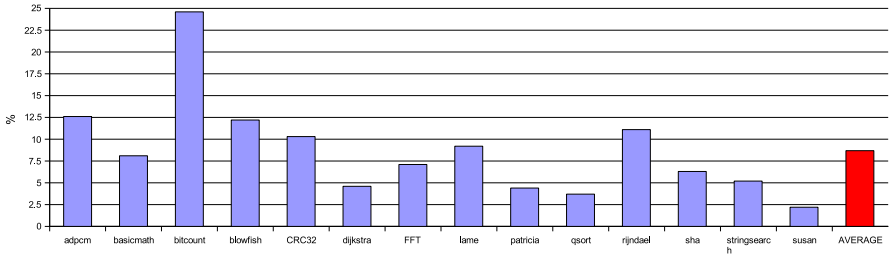

Fig. 7. Percentage of failed loops

5.3 Deactivation of the Loop Unrolling Optimization

Previous compilation phases can return an error result if the estimated unroll factor remains below a threshold, or the required registers inside a target loop cannot be fed by the unroll bank of registers. In those cases, provided that the main goal is power reduction, the loop unrolling optimization will be deactivated for the loop under consideration. When the estimated unrolling factor is below the previously set threshold, the optimization will be deactivated. Therefore, the banked approach previously presented can perform without modification and the energy savings will correspond to those unused register file banks.

5.4 Experimental Results

Figure 7 shows the percentage of failed loops (those that exploit the register demands and cannot be fed by the register file banks) over the total number of loops, and for some benchmarks selected from the MiBench suite. As can be seen, the behavior of every bechmark regarding the percentage of failed loops is quite different, ranging from the 2.2% of the *susan* benchmark, to the 24.6% of *bitcount*. The baseline architecture for these simulations corresponds to that described in section 4.

The loop unroller factor is estimated after a post-compilation stage. In this approach, the compiler is allowed to finish the compilation process and, after recovering the selected unroll factor, recompiled with the new selection. Figure 9 shows the percentage of failed loops after the unroll factor has been selected on the post-compilation estimation. As can be seen, those failed loops due to the wrong estimation of the unroller loop are corrected, and the average count of failed loops is decreased to 5.96%. However, the unroller factor cannot always be effectively tuned and some failed loops still remain.

Figure 9 also shows the percentage of failed loops after compilation using an unroll bank of 64 registers and targeting the most-time consumming loop in the benchmark. As can be seen, some of the benchmarks (those with a loop which represents a significant portion of the execution time) present in this approach a better behavior, while others (those where a significant loop cannot be found)

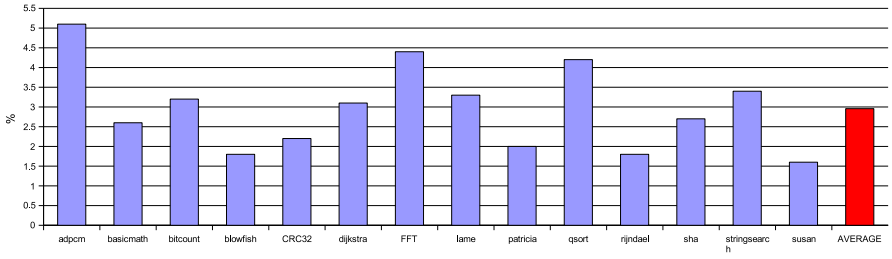

Fig. 8. Performance penalty after the use of the unroll bank of registers

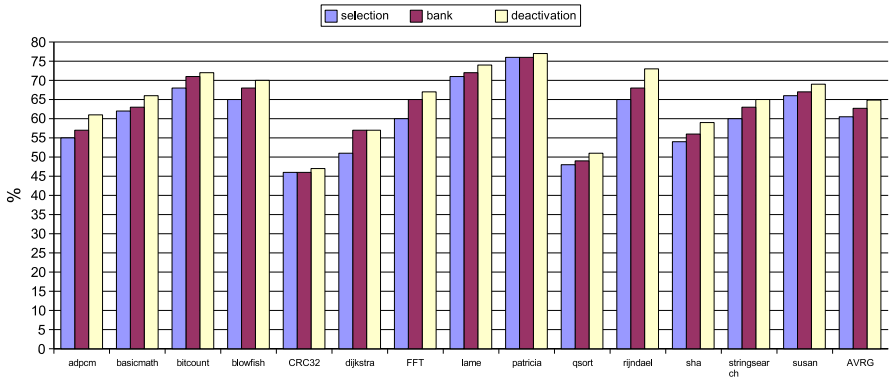

Fig. 9. Energy savings in the register file for the three unroller optimizations

have increased the percentage of failed loops. In average, the percentage of failed loops have been decreased to a 5.52%.

For this approach, the performance penalty incurred by the extra clock cycles needed to use the unroll bank of registers has also been measured (Figure 8). In average, this does not represent more than a 3%. There is also a performance penalty in terms of area of the die for the unroll bank. Usually, when targeting low-power, area of the chip can be sacrified in order to meet the energy constraints.

Some of the benchmarks' loops cannot be fit in the register file banks by tuning the loop unrolling factor and meeting at the same time the defined threshold. For those loops, the loop unrolling optimization is deactivated. The size of the register file banks (32 register) has been selected in the way that they can feed the register demands for every benchmarks when no unrolling is applied. Therefore, the percentage of failed loops after the deactivation of the unrolling is very reduced (Figure 9).

Finally, Figure 9 shows the energy savings in the register file for the three different approaches related to the loop unrolling optimization (selection of the unroll factor, unroll bank and deactivation of the optimization). In this graph, it

can be observed the improvement in terms of energy savings when simultaneously using all these approaches.

6 Ongoing Work: Extension to MPSoCs

The semiconductor industry is still facing several technological challenges to build the MPSoC systems. They require an enormous computational performance (2 - 30$GOPS$) with low energy consumption demands (0.3- 2W) [25]. Although current desktop processors offer these performance requirements, they consume too much power (10-100W) [26]. Therefore, while keeping the performance figures, the power consumption needs to be at least two or three orders of magnitude lower. Within this context, methods to reduce the power consumption of the new MPSoC platforms are in great need.

Analyzing current implementations of MPSoCs and the trend on the market, it can be said that the trends in future platforms are:

- Platforms with an increasing number of processing units, working in cooperation to perform the work;
- Heterogeneous architectures with different kind of processors, hardware accelerators and co-processors;
- Shared implementation of functional units for the optimal use of resources;
- Availability of low-power modes of operation;

6.1 Proposed Architectural Extensions

The work developed in this section has been performed with the use of CRISP [27]. CRISP has been developed at the Katholieke University of Leuven and IMEC, and consists of a high-level simulation platform of heterogenous architectures and its framework for the automatic generation of compilers.

The baseline architecture described by CRISP consists of a selectable number of processing elements (VLIW processors) which communicate with a shared register file through a full crossbar network (Figure 10). This architecture has been extended and modified in the following way to support the unrolling mechanisms proposed in this paper:

- The shared register file among all processing elements has been split into several banks which can be independently accessed by the processors. Then, a Dynamic Voltage Scaling (DVS) technique is applied to turn the unused banks into a low power state and thus save as much energy as possible in the system.
- Every processing element includes now an additional local register file with a reduced size compared to the shared register. All these local register files are switchted off during normal functioning.
- Hardware support is provided to the compiler to power up more than one bank of registers in the shared register file or local register files in several processing elements when they are needed. In normal execution of the system,

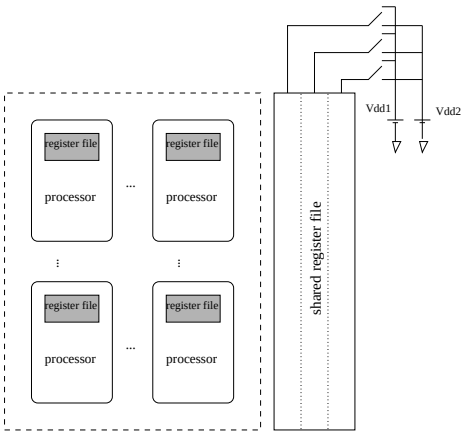

Fig. 10. CRISP architecture

most of the banks of the shared register file will be kept in a low-power state thanks to the modified register assignment implemented in the compiler. When needed, the register file banks or local register files will be powered up to feed the register demands of the code. The selection between both configurations (extra banks in the shared register file, or local register file) is based on energy considerations analyzed by the compiler using our proposed unrolling mechanisms for MPSoC systems (Section 5).

6.2 Benchmark Selection and Preliminary Results

The benchmarks used for the experimental work have been selected to analyze all the parameters that have a strong impact in the MPSoC behavior. These benchmarks are:

- MPEG-2: This video coding algorithm has been selected due to its low data dependency (the coding effort is not dependent on the video sequence), processing layer by layer, and high parallelization possibilities.
- JPEG2000: This image coding algorithm presents high data dependency, and high parallelization possibilities.

The preliminary simulations on this topic show how the results highly depend on the parallelization possibilities of the benchmark, as well as the data dependencies between the processors in the MPSoC. Those benchmarks where the algorithm can be independently split into several processors (with scarce or no inter-processor communication) are able to take advantage of the proposed approach. Moreover, the capability to detect the data dependencies between processors helps on deciding the best unrolling policy and the precise use of the unroll-bank.

7 Conclusions and Future Research

The reduction of complexity and energy consumption in the register file is a primary goal in the design of complex processor-based systems. This research work has shown how the use of power-aware compilers can provide significant savings in this device by a careful design of the underlying architecture.

This work presents different compiler-based mechanisms to target power and complexity optimizations in the register file of out-of-order processors. In particular, a reduction of the number of ports in the register file (which impacts on the energy consumption and design complexity) by means of a banked architecture and a specific register assignment is presented.

Also, several unrolling policies have been evaluated and proposed to reduce the energy consumption in the register file of single-processor and MPSoC architectures.

The ongoing research work carried out by our group is focused on the analysis of the impact of compiler optimizations in the power consumption of the register file, as well as the automatic tunning of these optimizations to improve the energy savings. Also, this work is currently being extended to work at different granularity levels (loop, function, etc), with heterogenous MPSoC architectures, and also targeting the cache hierarchy.

Acknowledgements

This work was supported by the Spanish Ministry of Science and Technology under contract TIC2003-07036.

The authors would like to thank Alex Veidenbaum (from University of California in Irvine), David Atienza (from Complutense University of Madrid) and Praveen Raghavan (from IMEC) for their support and constructive feedback in this work.

References

1. Gonzales, D.R.: Micro-RISC Architecture for the Wireless Market. In: International Symposium on Microarchitecture. (1999)
2. Folegnani, D., González, A.: Energy-Effective Issue Logic. In: International Symposium on Computer Architecture. (2001)
3. Mahlke, S.A., Chen, W.Y., Chang, P.P., Hwu, W.: Scalar Program Performance on Multiple-Instruction Issue Processors with a Limited Number of Registers. In: Hawaii International Conference on System Sciences. (1992) 34–44
4. Postiff, M., Greene, D., Mudge, T.: The Need for Large Register File in Integer Codes. Technical Report CSE-TR-434-00, Electrical Engineering and Computer Science Department. The University of Michigan (USA) (2000)
5. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.N.: Drowsy Caches: Simple Techniques for Reducing Leakage Power. In: International Symposium on Computer Architecture. (2002)

6. Steidl, S.A.: A 32-Word by 32-Bit Three-Port Bipolar Register File Implemented Using a SiGe HBT BiCMOS Technology. PhD thesis, Rensselaer Polytechnic Institute (2001)
7. Zyuban, V.V., Kogge, P.M.: The energy complexity of register files. In: International Symposium on Low Power Electronics and Design. (1998)
8. Seznec, A., Toullec, E., Rochecouste, O.: Register write specialization register read specialization: A path to complexity-effective wide-issue superscalar processors. In: MICRO. (2002)
9. Cruz, J.L., González, A., Valero, M., Topham, N.P.: Multiple-banked register file architecture. In: nternational Symposium on Computer Architecture. (2000)
10. Park, I., Powell, M.D., Vijaykumar, T.N.: Reducing register ports for higher speed and lower energy. In: MICRO. (2002)
11. Koen, J.P., Langendoen, K., Sips, H.J.: Application-directed voltage scaling. IEEE Transactions on Very Large Scale Integration (TVLSI) **11** (2003) 812 – 826
12. Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Ye, W.: Influence of compiler optimizations on system power. In: Design Automation Conference. (2000)
13. Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., Veidenbaum, A., Nicolau, A.: Architectural and compiler strategies for dynamic power management in the copper project. In: International Workshop on Innovative Architecture. (2001)
14. Parikh, A., Kim, S., Kandemir, M., Vijaykrishnan, N., Irwin, M.: Instruction scheduling for low power. Journal of VLSI Signal Processing (2004) 129–149
15. Steinke, S., Schwarz, R., Wehmeyer, L., Marwedel, P.: Low power code generation for a RISC processor by register pipelining. Technical report, Dept. Comput. Sci. XII, Univ. Dortmund, Dortmund, Germany (2001)
16. Akturan, C., Jacome, M.F.: FDRA: A software-pipelining algorithm for embedded VLIW processors. In: Proceedings of ISSS. (2000) 34–40
17. Ayala, J.L., López-Vallejo, M., Veidenbaum, A.: Energy-efficient register renaming in high-performance processors. In: Proceedings of WASP. (2003)
18. Ayala, J.L., López-Vallejo, M.: Improving register file banking with a power-aware unroller. In: Proceedings of PARC. (2004)
19. Zyuban, V.V., Kogge, P.M.: The energy complexity of register files. In: Proceedings of ISLPED. (1998)
20. Reinman, G., Jouppi, N.: An integrated cache timing and power model. Technical report, COMPAQWestern Research Lab (1999)
21. Ayala, J.L., López-Vallejo, M., Veidenbaum, A.: Power-aware register renaming in high-performance processors using compiler support. In: Proceedings of IWIA. (2004)
22. Farkas, K.I., Jouppi, N.P., Chow, P.: Register file design considerations in dynamically scheduled processors. In: International Symposium on High Performance Computer Architecture. (1996)
23. Janssen, J., Corporaal, H.: Partitioned register file for TTAs. In: International Symposium on Microarchitecture. (1995)
24. Seng, J.S., Tullsen, D.M.: The effect of compiler optimizations on Pentium 4 power consumption. In: Workshop on Interaction between Compilers and Computer Architectures. (2003)
25. Viredaz, M., Wallacha, D.: Power evaluation of a handheld computer. IEEE Micro **23** (2003) 66–74
26. Bose, P., et al.: Early-stage definition of lpx: A low power issue-execute processor. In: Proceedings of PACS. (2002)

27. Barat, F., Jayapala, M., Aa, T.V., Lauwereins, R., Deconinck, G., Corporaal, H.: Low power coarse-grained reconfigurable instruction set processor. In: 3th International Conference on Field Programmable Logic and Applications, 1st - 3rd Sept. 2003, in Lisbon, Portugal (2003)